

---

Research

# Visualizing Feature Evolution of Large-Scale Software based on Problem and Modification Report Data



Michael Fischer<sup>1,†</sup> and Harald Gall<sup>2,‡,\*</sup>

<sup>1</sup>*Distributed Systems Group, Vienna University of Technology, Argentinierstrasse 8/184-1, A-1040 Vienna, Austria*

<sup>2</sup>*Department of Informatics, University of Zurich, Winterthurerstrasse 190, CH-8057 Zurich, Switzerland*

---

## SUMMARY

Gaining higher level evolutionary information about large software systems is a key challenge in dealing with increasing complexity and architectural deterioration. Modification reports and problem reports taken from systems such as CVS and Bugzilla contain an overwhelming amount of information about the reasons and effects of particular changes. Such reports can be analyzed to provide a clearer picture about the problems concerning a particular feature or a set of features. Hidden dependencies of structurally unrelated but over time logically coupled files exhibit a high potential to illustrate feature evolution and possible architectural deterioration. In this paper, we describe the visualization of feature evolution by taking advantage of this logical coupling introduced by changes required to fix a reported problem. We compute the proximity of problem reports by applying a standard technique called Multidimensional Scaling (MDS). The visualization of these data enables to depict feature evolution by projecting problem report dependence onto (a) feature-connected files and (b) onto the project directory structure of the software system. These two different views show how problem reports, features and directory tree structure relate. As a result, our approach uncovers hidden dependencies between features and presents them in easy-to-assess visual form. A visualization of interwoven features can indicate locations of design erosion in the architectural evolution of a software system. As a case study, we used Mozilla and its CVS and Bugzilla data to show the applicability and effectiveness of our approach.

KEY WORDS: software evolution, problem reports, release history, visualization

---

\*Correspondence to: Harald Gall, Department of Informatics, University of Zurich, Winterthurerstrasse 190, CH-8057 Zurich, Switzerland

<sup>†</sup>E-mail: [fischer@infosys.tuwien.ac.at](mailto:fischer@infosys.tuwien.ac.at)

<sup>‡</sup>E-mail: [gall@ifi.unizh.ch](mailto:gall@ifi.unizh.ch)

Contract/grant sponsor: Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT), The Austrian Industrial Research Promotion Fund (FFF), and the European Commission; contract/grant number: EUREKA 2023/ITEA projects CAFE and FAMILIES



---

## 1. INTRODUCTION

Analyzing software evolution can be addressed from many different viewpoints. We focus on analyzing the evolution of features since they are a natural unit to describe software from the perspective of the application user and the software developer. A software design may erode during the project's lifetime and features which were initially implemented in different modules become more and more interwoven. Our goal is to reveal this architectural deterioration through the use of information collected during implementation and maintenance. This facilitates the comprehension of a software system in the light of a complex reengineering process [1].

Version control systems such as CVS or ClearCase and bug tracking systems such as Bugzilla or ClearQuest are information repositories that contain a tremendous amount of historical information but in most cases are not further exploited and only used as a passive data container. But several studies [2, 3, 4, 5, 6] have shown the potential of these data to carry meaningful historical information for many kinds of software evolution analyses.

By analyzing problem and modification data, our approach reveals how and where in the source tree features reside, how they have co-evolved over time, and to which degree they are coupled to each other. Distances between features can be expressed as the (inverse) number of problem reports that two features have in common. As a result, dependencies such as logical coupling through problem reports can be used to group features to sets of closely related files and classes.

Architectural deterioration can be identified, for example, if a single problem report requires the fixing of many files in different parts of the system and if bug fixing of this kind of problem almost always involves the same set of files to be changed. Or, if a feature is implemented in many different parts of the system with increasing scattering of its implementation across many system parts over time. Feature co-evolution and feature dependencies, on the other hand, can be identified, for example, if in most cases two (or more) features are changed together based on the same (kind of) problem report. All these evolution analysis data can be visualized by an automatic layout algorithm that creates graphical representations to be used by a software engineer as diagnostic aid in detecting design erosion or architectural deterioration.

We have taken Mozilla as a case study to show the effectiveness of our approach. Mozilla consists of about 10.480 C/C++ source files comprising 3.7 millions of source lines in 2.500 subdirectories. The bug tracking system Bugzilla contains more than 180.000 problem reports and the version control system CVS contains more than 430.000 modification reports. Mozilla basically consists of 90 modules maintained by 50 different module owners. Given these characteristics, we consider Mozilla and its historical data as a large open-source software system that has been developed quite successfully over several years.

*Modification Reports* (MRs) taken from Mozilla's CVS repository and *Problem Reports* (PRs) taken from the Bugzilla bug tracking repository contain an overwhelming amount of information about the reasons for small or large changes to the software, for which adequate data filtering mechanisms need to be applied to enable useful and meaningful analyses. Hidden dependencies of structurally unrelated but over time logically coupled files (i.e., files that most often are changed together although residing in separate modules or subsystems) further exhibit potential to illustrate feature evolution and possible architectural shortcomings.

The approach presented here addresses this problem by grouping feature related PRs and presenting the results in visual form. The input data for this process are selected from a *Release History Database*

---



(RHDB) [7] which contains MRs, PRs and feature data. For every feature, which shall be inspected, the related PR information is selected from the RHDB. Then, the distance between two PRs can be expressed as the number of files commonly modified to fix both problems. The more files they have in common the closer is the distance between the PRs. On the basis of these proximity data, groups of related reports are formed by applying a technique called *Multidimensional Scaling* (MDS) [8] on this data. Results of the MDS process are visualized in a two (or optionally higher) dimensional space.

Grouping of PRs can reveal hidden dependencies between features but can be also used to identify groups of commonly modified program code. Results from this analysis can be used as indication for a degraded system architecture or as indication of design erosion.

The remainder of this paper is organized as follows: Section 2 gives a brief overview of related work in the area of visualization of large-scale software evolution. In Section 3 we introduce the data sources used, the selection and qualification of relevant problem reports as well as the feature extraction process. Section 4 describes the visualization of the feature evolution resulting in two different kinds of views: feature view and project view. We conclude the paper in Section 5 and indicate areas for future research.

## 2. RELATED WORK

The analysis of software evolution based on release history information and the visualization of these results has been addressed by only a few researchers so far. Some of them also have used Mozilla as a case study for their investigations.

In [9] Taylor and Munro describe an approach based on revision data to visualize aspects of large software such as active areas, changes made, or sharing of workspace between developers across a project by using animated revision towers and detail views. Since their approach is purely based on revision history, additional important information such as problem reports or feature data are not considered for visualization.

Similar to our environment used to produce the results presented in this paper, Draheim and Pekacki propose a framework for accessing and processing revision data via predefined queries and interfaces [10]. Linkage of their data model with other evolutionary project information—such as problem report data as required for our analysis—and making them accessible for external queries is beyond the scope of their work.

Mozilla has been already addressed, for example, by Mockus, Fielding, and Herbsleb in a case-study about Open Source Software projects [11]. They also used data from CVS and the Bugzilla bug tracking system but—in contrast to our work—focused on the overall team and development process such as code contribution, problem reporting, code ownership, and code quality including defect density in final programs, and problem resolution capacity as well.

Bieman et al. [12] used change log information of a small program to detect change-prone classes in object oriented software. The focus was on visualizing classes which experience common frequent changes, which they called *pair change coupling*. Instead of grouping coupled objects they used standard UML diagrams together with a graph showing the number of pair change couplings between change-prone classes to visualize their analysis results.

Similar to an earlier approach of our group described in [13], Kemerer and Slaughter used modification reports as basis for their analysis [5]. They used a refined classification scheme for



modification reports (corrective, adaptive, perfective enhancement [14], and new program) for an analysis of ordered change events and put quite some effort in the classification of change events. As a result, they were able to reveal different phases of a system's life cycle. Unfortunately, formal mechanisms to record such historical data during the development process are still not supported in software development tools. While they have thoroughly investigated longitudinal system evolution, our focus is on visualizing hidden dependencies between components of a system reflected by any kind of traceable pattern, e.g., commonly and frequently changed modules or common problem reports.

In [15, 16] Lanza depicts several releases of a software system in a matrix view using rectangles. Width and height of rectangles represent specific metrics (e.g. number of methods, number of instance variables of classes) according to the history of classes are visualized. Based on this generated evolution matrix, classes are assigned to different evolution categories such as, for example, pulsar (class grows and shrinks repeatedly) or supernova (size of class suddenly explodes). Whereas he analyzes the evolution of classes we focus on features. However, a combination of both approaches could be promising.

In [13, 17] our group examined the structure of a *Telecommunications Switching Software* (TSS) over more than 20 releases to identify logical coupling between system and subsystems; a similar study has been carried out by Bieman et al. in [12]. Based on release history data of this TSS, Riva et al. presented an approach to use color and 3D to visualize the evolution history of large software systems [18]. Colors were primarily used to highlight main incidents of the system's evolution and reveal unstable parts of the system. In the interactive 3D presentation it is possible to navigate through the structure of the system on a very coarse level and inspect several releases of the software system. Our work extends this approach by including feature and problem report data.

Zimmermann, Diehl and Zeller presented a fine-grained analysis approach for CVS data that considered all kinds of entities starting from the statement level [19]. Their ROSE prototype identifies common changes between syntactical entities rather than files or modules, which are the focus in our work. In their work they describe the clear limitations of such an approach for fine-grained evolution analysis.

Dickinson et al. [20] used multidimensional scaling to cluster execution traces of faulty programs and compare the output with pre-determined test-cases. Our approach differs in that we operate on source level and problem reports indicated on features and source level elements.

### 3. BUILDING UP AND FILTERING QUALIFIED EVOLUTION DATA

To visualize evolution it is first necessary to build up a relevant set of data by filtering and qualifying data across information sources. Our analysis is based on three main sources: (1) modification reports (MRs) taken from the *Concurrent Versions System* (CVS) [21]; (2) problem reports (PRs) taken from Bugzilla [22]; and (3) the executable program to extract feature information. Data in sufficient quantity and quality is offered via publicly accessible resources from the Mozilla [23] project.

The Bugzilla bug tracking system supports seven severity levels of problem reports [24]: *blocker* ("application unusable," 4.460 PRs), *critical* ("crashes, loss of data, severe memory leak," 17.946 PRs), *major* ("major loss of function," 21.041 PRs), *normal* (10.139 PRs), *minor* ("minor loss of function, or other problem where easy workaround is present," 112.380 PRs), *trivial* ("minor cosmetic issue," 3.992 PRs), and *enhancement* (13.349 PRs). Since the Bugzilla bug tracking system already offers quite well

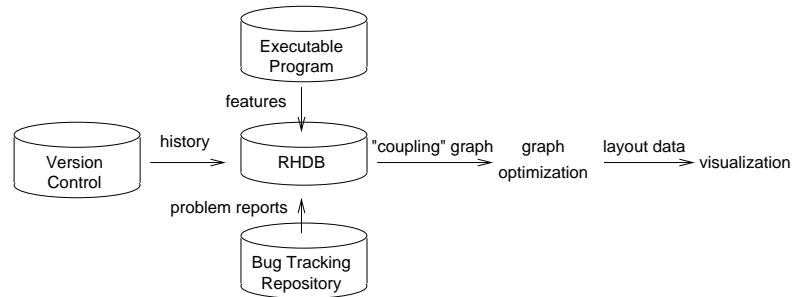


Figure 1. Information flow and process steps

classified PR data in sufficient quantity and quality, we decided to stay with the existing scheme to identify coupling between system parts. On a coarse level Mozilla can be divided into 90 modules maintained by 50 different module owners. For the current Mozilla application suite (version *1.3a*) there exist 10.477 source files in C/C++ containing 3.731.552 source text lines or 2.443.085 lines of code in 10.434 files (see also [4]). For the selected components in our case study we counted 733.573 source text lines or 515.529 lines of code. Given these key data, Mozilla can be classified as a large scale software project with solid software evolution analysis challenges.

### 3.1. The process of analyzing evolution data

The information flow and the process steps for analyzing evolution data is depicted in Figure 1. Data are extracted as follows: the CVS is used to retrieve the historical data provided as MRs; the executable program is required to gain feature information; and from the bug tracking system the problem report information is retrieved. From these three sources the *Release History Database* (RHDB) is built up which provides a filtered, validated and thus qualified set of data for our purpose. Queries are used to retrieve the required set of input data for the optimization process from the RHDB. The output of this step is used for visualization. The data sources and the process steps are described in more detail in the following sections.

### 3.2. The Release History Database (RHDB)

We have downloaded the relevant MRs and PRs, filtered, validated, and stored the data in our RHDB according to our data filtering mechanisms described in [2, 7]. General information about entities and artifacts of the software's source code and modification reports are stored in the RHDB. The relation between them was rather easy to establish, since the relevant information is contained in consistent form within the change logs from the CVS repository. PRs from the bug tracking system are stored as separate entities. Crucial in the reconstruction of the RHDB was the re-establishment of the linkage between MRs and PRs since no formal mechanisms are provided by CVS to link this information. In



the rebuild process we used the problem report IDs found in the modification reports of CVS to link modification reports and problem reports. These IDs have been entered by the authors of source code modifications as free text. Natural problems were: (1) context not clear in which an ID was used; (2) incorrect report IDs, e.g., typos; or (3) no IDs were identified at all. Whereas we were able to solve the first two problems with reasonable effort (e.g., fine grained regular expressions, exclusion of number ranges), we did not find a feasible solution for the third problem to reconstruct this information, e.g., by evaluation of patch data attached to problem reports.

IDs in MRs are detected by a set of regular expressions. A match is rated according to the confidence value we have assigned to the expression and can be *high*(h), *medium*(m), or *low*(l). The confidence is considered high if expressions such as `<keyword><ID>` can be detected whereas a five digit number just appearing somewhere in the text of a MR without preceding keyword is considered low.

To verify the results from the reconstruction process we used patch information which is sometimes attached to problem reports. These patches contain file name information which can be used to validate the results from the linkage reconstruction. If a patch was found which confirms the linkage, the rating value was changed from (h) to (H), (m) to (M) and (l) to (L), respectively. Details can be found in [2].

### 3.3. Qualification and selection of problem reports

Next, we present our quantitative results for reconstructing and validating links between MRs and PRs. Table I compares the data for the most important PR categories “resolved” and “verified” of the Bugzilla database. Column “all” lists the total amount of reports downloaded, whereas “ref” lists the number of references found in MRs.

Table I. Problem reports from Bugzilla

\ Status Resolution	resolved	verified
	all / ref	all / ref
duplicate	13855 / 66	41648 / 487
fixed	14806 / <b>7705</b>	36620 / <b>18940</b>
invalid	4551 / 43	9116 / 113
won't fix	1823 / 46	3123 / 69
works for me	9765 / 93	15569 / 308

Table II. Products and problem reports

Product	fixed reports			all reports		
	all	found	%	all	found	%
Browser	35520	20396	57.42	129889	22208	17.10
Bugzilla	1630	9	0.55	4564	26	0.57
MailNews	7089	4583	64.65	29112	4978	17.10
Tech Evangelism	1314	6	0.46	4908	16	0.33
mozilla.org	1126	5	0.44	2122	15	0.71

Table II gives a summary of PRs for some products covered by the Bugzilla database. For the product we are interested in, i.e., “Browser” or “MailNews” which is part of the Mozilla application suite, we found at least 50% of the downloaded reports. Indicative for a false positive detection (or maybe wrong tracking state) are the categories “Bugzilla”, “Tech Evangelism,” or “mozilla.org” since they cover no browser development related issues.

To further improve the results of our analysis process, we need to reduce the impact of PRs not directly related to fixing a specific functional problem. We inspected the description of the largest reports and tried to find a criterion for the selection of our data sets. Reports such as “license foo” (PR-ID #98089, with 7.961 referenced files), “printfs and console window info needs to be boiled away for release builds” (#47207, 1.135), or “Clean up SDK includes” (#166917, 888) are considered as irrelevant since they primarily concern administrative problems. Since MRs such as “libtimer\_gtk\_s is causing link problems” (#11159, 300) or “repackage resources into jar files” (#18433, 289) are still concerned with administrative issues, we decided to use 255 as the upper bound of referenced files per



Table III. Scenario definitions with features

Scenario	Description	Feature	Fill style	ID	Files
Core	Mozilla start / blank window / stop	Core		00	705
HTTP	TrustCenter.de via HTTP*	Http		01	28
HTTPS	TrustCenter.de via SSL/HTTP	Https		02	6
File	read TrustCenter.de from file	-		-	-
XML	XML Base (see <a href="http://www.w3c.org/">http://www.w3c.org/</a> )*	Xml		09	65
MathML	mathematic in Web pages*	MathML		08	13
About	“about:” protocol	About		10	3
fBlank	read blank html page <sup>†</sup>	Html		03	76
hBlank	blank html page via HTTP*	-		-	-
Image	-	Image		04	3
ChromeGIF	Mozilla logo: <a href="chrome://global/content/logo.gif">chrome://global/content/logo.gif</a>	ImageGIF		07	4
PNG	image: Portable Network Graphics*	ImagePNG		05	10
JPG	image: Joint Photographic Experts Group*	ImageJPG		06	16

MR to define a relevant problem report for our considerations. This heuristic defines a relevant problem report for our considerations and guarantees not to lose any “major” or “critical” PRs.

### 3.4. Feature extraction

Since features are used in communication between users and developers it is important to know which features are affected by (future) functional modifications of a software system. According to [25] a feature is *a prominent or distinctive aspect, quality, or characteristic of a software system or systems*. For our purposes we use the more practical definition of a feature as *an observable and relatively closed behavior or characteristic of a software part* [26].

The goal of the feature extraction process is to gain the necessary information to map the abstract concept of features onto a concrete set of files which implement a certain feature. To extract the required feature data we applied the software reconnaissance technique [27, 28] within our *Linux (RedHat 8.0 & SuSE 8.1)* development environment. GNU tools [29] have been used successfully in [30] to extract feature data.

First, we created a single statically linked version of Mozilla with profiling support enabled. From several test-runs where the defined scenarios (see Table III) were executed, we created the call graph information using the GNU profiler. The call graph information was used to retrieve all functions and methods visited during the execution of a single scenario. Since our analysis process works on the

\*copies of the Web pages used are available at <http://www.infosys.tuwien.ac.at/staff/mf/jsme/elisa/>

<sup>†</sup>the file contains the following HTML tags:<html><head></head><body></body></html>

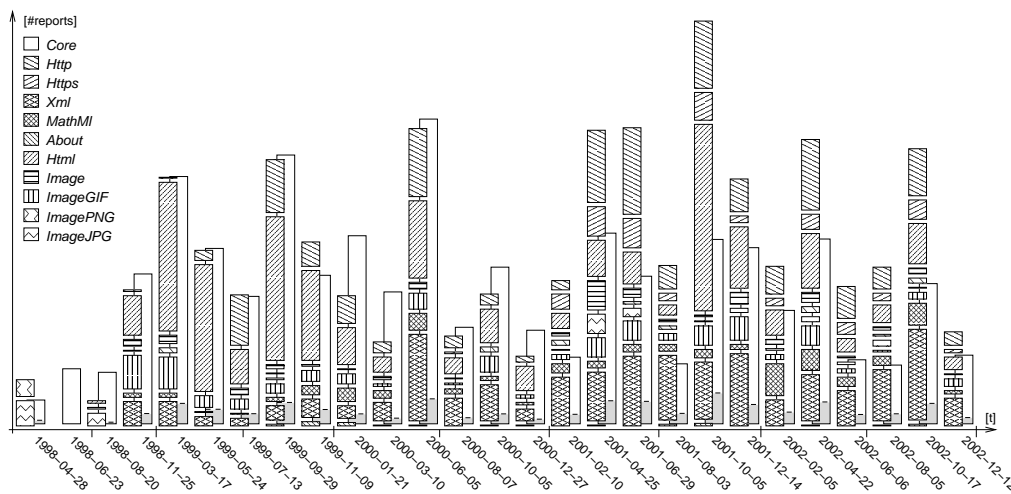


Figure 2. Distribution of problem reports fixed for core and selected features

file level, we mapped function and method names onto a higher level of abstraction. In the next step, “feature data” were extracted from file name mappings using set operations. For example, the *Xml* feature was extracted by the following expression:

$$\text{Xml} = (\text{MathML} \cap \text{XML}) \setminus (\text{Core} \cup \text{HTTP} \cup \text{PNG} \cup \text{fBlank} \cup \text{hBlank} \cup \text{ChromeGIF})$$

The names in the above expression represent the set of files extracted in the previous steps from the executed scenarios. Table III also lists the names assigned to the features, the file styles which are used for their visualization, and the number of files retrieved. Finally, we imported the filename-information into the RHDB along with the release number of the program from which the data were retrieved. In our case it was Mozilla version 1.3a with the official freeze date 2002-12-10 (even though we found one MR with a time-stamp 2002-12-12).

The distribution for all types of reports referenced from MRs for the *core* feature (represented as white bars in the background) and the other extracted features (represented by different fill styles) is depicted in Figure 2 using a bi-monthly time-scale. Since the number of reports for the *core* is a magnitude larger than the number of reports for the other features, we used a scale ratio of 10 : 1 for the boxes. The largest number of reports found for a single period was 3.628 ending on 2000-06-05. To visualize the *core*-to-feature ratio, the bottom of the white bars are shaded according to the ratio calculated.

It can be seen from the number of fixed PRs for the features, that periods with less activities are during the summer time and at the end of each year. Thus we decided to use one year as time-frame in the visualization of feature dependencies in Figure 3.



Table IV. Total number of ‘couplings’ between features

Feature	Feature										Period				Total
	01	02	03	04	05	06	07	08	09	10	<2000	2000	2001	2002	
01/Http	0	20	10	7	0	2	2	0	7	4	24	63	155	129	371
02/Https		0	2	1	0	0	0	0	2	0	0	1	69	61	131
03/Html			0	6	0	1	46	16	29	1	87	116	140	122	465
04/Image				0	0	15	5	0	3	2	5	5	54	31	95
05/ImagePNG					0	0	0	0	0	0	0	0	3	5	8
06/ImageJPG						0	0	0	0	0	1	0	22	11	34
07/ImageGIF							0	17	36	0	15	61	75	38	189
08/MathML								0	67	0	2	9	32	72	115
09/XML									0	1	7	34	105	110	256
10/About										0	2	1	2	1	6

#### 4. VISUALIZING FEATURE EVOLUTION

Visualization is a useful technique to present complex feature interrelationships. We use two different types of views to facilitate the understanding of evolutionary changes in large software systems: (1) the *feature-view* focuses on the problem report based coupling between the selected features; and (2) the *project-view* depicts the reflection of problem reports onto the directory structure of the project-tree.

##### 4.1. Feature view: projecting PRs onto features

This view focuses on the visualization of features and their dependencies via problem reports. The degree of coupling between two features is represented by edges whereas the number of references (i.e., the edge weight) from PRs to files is expressed as line-width. Each line indicates the coupling of files through PRs on the feature level rather than the file level. In fact, all entities contributing to a feature are drawn on the same position, which supports the impression that features are compared.

To reduce the amount of visible edges—one problem report can affect several items of a feature—and to visualize only important edges, we use the following criteria: (1) the edge weight between nodes is set to 200—if the actual weight is greater—to reduce the impact of outliers, i.e., dominant edges (upper bound); and (2) an edge must have at least 10% of the weight of the highest weighted edge to be visible (lower bound). As a consequence, every edge in Figures 3.b, 3.c and 3.d represent at least 20 references. Since the maximum number of references in Figure 3.a is 168, the smallest visible edges represent 17 references. To scale down from the large number of references we used the logarithm function to determine the visual line-width. The actual number of PRs that features have in common and the total number of PRs for every feature can be found in Table IV. Due to the small number of files—which means a small number of PRs—we identified for the features *ImageGIF*, *Image*, and *About* and the threshold for references, almost no coupling is depicted even though some references do exist. If a feature is not shown in one of the figures, we could not find any PR for the specified period or the threshold was not reached. An isolated feature indicates only “local” PRs but no references to other features.

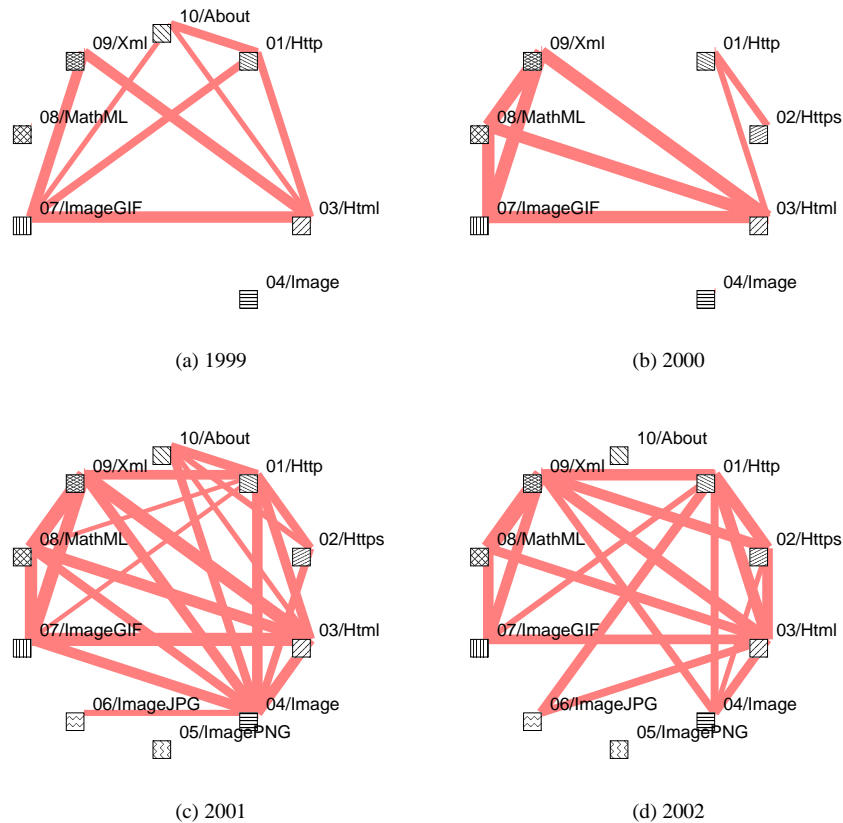


Figure 3. Dependencies between features reflected by large problem reports

Figure 3 depicts the results for the observation periods 1999, 2000, 2001, and 2002, in which features are aligned on a circle and visualized according to the fill style given in Table III. From 1998 till 1999—the result is depicted in Figure 3.a—we found virtually no coupling between features. The result changed in the subsequent observation periods not dramatically but constantly. In Figure 3.b the situation for 2000 is depicted and indicates that the focus shifted from *Http* to other features such as *Html*, *Xml*, and *MathML*.

In 2001, the situation changed substantially and is depicted in Figure 3.c. The partially connected graph has turned into an almost fully connected graph and the number of reported and fixed problems have more than doubled (from 290 to 657). Except for feature *ImagePNG*, all features are affected



by system-wide changes! In 2002 (see Figure 3.d) the situation slightly improved since the number of reported problems dropped to 580.

As a result, this kind of visualization can be used to point out phases of architectural deterioration on the basis of feature dependencies. Our window of observation for every figure was one year but that can be changed according to the given data set of the particular system under study. Such visualizations can be used (a) to assess the point of time when some restructuring or reengineering activities should be started and (b) to estimate the likely amount of resources to be required for changing particular features and/or system parts.

## 4.2. Project view: projecting PRs onto project-tree structure

The goal of this view is to visualize the reflection of PRs onto the directory structure of the project tree. We assign weights to both the edges of the tree structure and the edges introduced through the coupling of PRs, and to search for groups in the resulting data set. The output of the optimization process, i.e., the node positions of the optimized graph, is visualized using a conventional drawing program, whereas the resulting graph is enriched with feature information.

Formally, two nodes  $v_i$  and  $v_j$  of the graph  $G$  are connected if a directory path of the project tree exists between these nodes such that we can define a relationship  $R$  in the form  $v_i R v_j$ , or the nodes share a problem report  $p_n$ , denoted as  $v_i R p_n \wedge v_j R p_n$ . The weight for an edge between the nodes are computed by Formula (1), in which  $n$  specifies the current number of connections between the two nodes and  $n_{max}$  the (global) maximum number of connections between any two nodes of the graph.

$$\text{weight}(v_i, v_j) = (-1) \left( \frac{n}{n_{max}} \right)^k + o \quad (1)$$

For an offset  $o = 1$  all weights are mapped by the above formula onto a range of  $[0..1]$  where 0 means the closest distance ( $k$  controls the distance generation between closely related nodes). We use the tool *Xgvis* [31]—it implements the multidimensional scaling algorithm—to process the graph to obtain the final position information of the graph nodes. The position information is then used as input for the drawing program *Xfig* [32].

### 4.2.1. Generating the visualizations

Input data for *Xgvis* and *Xfig* used for visualization are generated by a Java program. This program accepts some arguments which allow the user to control the data selection and generation process (date-range, features, PR severity). A critical step in the data generation process is the selection of parameters for the weights since this has a direct impact on the final layout. We used a ratio of  $\geq 20 : 1$  for project-tree edges and PR edges. This scheme gives more emphasis on the directory structure than to connections introduced by PRs. In a first phase of the data generation process, the objects of the project tree are assigned to their respective nodes of the graph. The minimum child size (*minchild*) specifies which nodes remain expanded or will be collapsed. Collapsing means that objects from the sub-tree are moved up to the next higher level until the size criterion is met, but not higher than the first level below the “ROOT” node. Since the complete Mozilla project tree consists of more than 2.500 subdirectories, we had to simplify the resulting graph and so we cut off unreferenced directories. In a second step it is possible to move fewer referenced nodes to a higher level to obtain a more compact



representation. The effect on the graph is that unreferenced leafs are suppressed, though they contain enough objects to meet the *minchild* criterion.

In Figures 4 and 5 the project directory tree is shown as gray nodes connected by black dashed lines. The root node is labeled “ROOT” and the features are indicated by filled in boxes according to the fill styles given in Table III. Coupling between nodes as result of common PRs are indicated by gray lines. Thicker lines and a darker coloring means that the number of PRs that two nodes have in common is higher. Since the optimization algorithm tries to place connected nodes close to each other, stronger dependencies can be spotted easily.

One marginal problem is the limited layout area in the two dimensional solution space: all nodes must be placed at least somewhere within a single plane and the placement of nodes after the optimization is only indicative within a certain radius. Naturally, this radius depends on the total number of nodes. By zooming-in, it is possible to provide a better picture of otherwise overlaid areas. An n-dimensional solution space could yield better results but it is very difficult to visualize. In general, the layout after optimization is one possible solution. It also does not necessarily mean that a global minimum for the given distances has been achieved. *Xgvis* supports up to 12 dimensions, which would yield better results for the optimization step, if, for example, more features were used; but then the results are difficult to visualize.

#### 4.2.2. Results: How features *Http*, *Https*, and *Html* relate

The three features *Http*, *Https*, and *Html* are depicted in Figure 4. As raw data we selected all PRs for these features with the exception of PRs classified as *enhancements* from the start of the project until the freeze date. Parameters, which influenced the project tree node selection process, were *minchild* = 10 (the number of artifacts, i.e. files in a subtree) and *compact* = 1 (the number of PR referenced by a node).

For the optimization process, we weighted the edges of the project tree with 20 (this gives more emphasis on the project structure), whereas an edge introduced through a single PR was weighted with 1. The factors  $k = 0.2$  and  $o = 0.2$  for the weight function were used to emphasize the spreading between nodes for visualization purposes.

The overall amount of PRs detected for a node is indicated via the diameter of a node and features “hosted” by a node are attached as boxes. Easy to recognize is the placement of nodes belonging to the *Html* feature on the right side, and *Http*, *Https* on the opposite side of Figure 4. Interesting are the nodes *netwerk.base*, *netwerk.protocol.http*, and *security.manager.ssl* since they are coupled via 90 PRs for *base* – *http* and 40 PRs for each of the other two edges. This indicates a high degree of coupling between the features *HTTP* and *HTTPS*.

Another interesting aspect is the spreading of the *HTML* feature over 10 different nodes. Modifications may be hard to track since several files in different directories contribute to a single feature. Remarkable are the two nodes *content.base* and *layout.html.base*, since they are coupled via 35 PRs although only 4 and 3 files are located in their respective directories.

As a result, Figure 4 shows strong change dependencies for the involved features across different directories and points to architectural deterioration distilled from the evolution data: (1) dependencies across branches of the project tree may indicate that there exist dependencies such as invocation, access, inheritance, or association; (2) frequently reported problems concerning certain source code locations of features can point to implementation or design problems; (3) features spread across the project tree

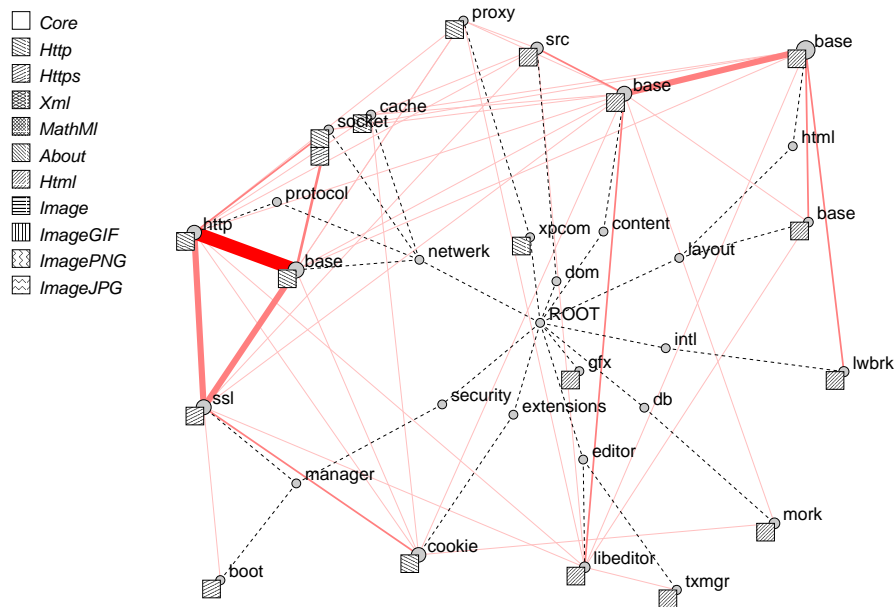


Figure 4. Relation of features *Http*, *Https*, and *Html* via problem reports

may indicate the involvement of large amounts of the code base for their realization and thus the impact of modifications can be difficult to predict.

#### 4.2.3. Results: How all features relate with the core

The *core* feature and all other investigated features are depicted in Figure 5 on a coarse grained level (edges that represent less than 5 references are omitted). For this configuration we selected all reports which were rated *major* or *critical*. We also set the minimum sub-tree size to 250 (*minchild*) entities and the minimum number of PR references to 50 (*compact*). This resulted in a graph with 37 nodes and 315 edges induced by PRs. By changing the values for *minchild* and *compact* it is possible to generate an arbitrarily detailed graph of the whole project. Since Mozilla has more than 2.500 subdirectories a complete graph representation of the whole project tree is far beyond the illustration capabilities of this medium. It is intuitive that the most critical subsystems in Mozilla are connected in the visualization, which is also supported by our findings.

The nodes with the highest density in *severe* PRs are *content* (with 608 references), *layout.html* (444), *layout.xul.base* (223), and *layout* (212). Another interesting aspect is the spreading of edges. In total 343 connections between nodes are depicted. If we select only those edges representing  $\geq 10$  references, then we can find the following ranking: node *content* with 19 edges, *layout.html* 10,





Table V. Frequently modified file pairs (subset)

P	M	Directory <i>content</i>	Directory <i>layout</i>
50	27	<i>events.src.nsEventManager.cpp</i>	<i>html.base.src.nsPresShell.cpp</i>
46	11	<i>base.src.nsDocumentViewer.cpp</i>	<i>html.base.src.nsPresShell.cpp</i>
23	13	<i>xul.content.src.nsXULElement.cpp</i>	<i>html.base.src.nsPresShell.cpp</i>
21	8	<i>xul.content.src.nsXULElement.h</i>	<i>html.style.src.nsCSSFrameConstructor.cpp</i>

and call graph information. Table V lists some of the topmost modified file-pairs of the two directories *content* and *layout*. Column “P” lists the number of MRs with associated PRs and column “M” lists the number of MRs without associated PRs. As the table indicates we can expect to find a strong relationship between event management (key, mouse, focus, etc.) located in directory *event*, the user interface components in *xul* (*XML User Interface Language*), and the visualization of HTML-content in *html*.

First, we show the dependency between *nsEventManager.cpp* (5384 source lines) and *nsPresShell.cpp* (7961 source lines) as listed in the first row of Table V. Since both files were frequently and pair-wise modified, we can expect to find some evidence in one of the source code deltas from CVS. In revision 1.302 of *nsEventManager.cpp* a call to function *FindContentForShell()* has been added as new code segment. The corresponding delta of *nsPresShell.cpp* (revision 3.462) reveals that the function has been introduced with this revision. Thus, we have found a new call relationship between these two files.

Second, we investigated revision 1.122 of *nsXULElement.h* (673 source lines) and revision 1.766 of *nsCSSFrameConstructor.cpp* (14330 source lines). An object type was modified; this modification—usually numerous files have to be modified if a type is changed—is also reflected in the number of affected files: 71 files in directory *content*, 23 in *layout*, 4 in *extensions*, and 2 in *xpfe*. From the source code deltas it is not possible to find a direct relationship, since only those source lines were modified where the data type was used in a declaration. An inspection of the call graph information from the feature extraction process reveals that the method *AttributeChanged()* of *nsCSSFrameConstructor.cpp* calls *GetMappedAttributeImpact()* of *nsXULElement.cpp* (5444 source lines). The include file *nsXULElement.h* declares this method which takes an argument of the modified type.

## 5. CONCLUSIONS AND FUTURE WORK

The graphical representation of dependencies between features based on problem report data opens up a new perspective on the evolution of software systems through retrospective analysis and visualization. By intuition problem reports should have a minimal impact on different features. Situations where this is not the case can be spotted easily through the graphical representation, e.g., in case of feature overlapping or spreading. We have applied multidimensional scaling (MDS) of problem reports linked with files and directory structures for the visualization of feature evolution of Mozilla for the years 1999 until 2002. The tool that we developed allows an engineer to generate two specific views of relationships and dependencies of a large software system: (1) the *feature view* provides a projection of problem reports onto the files that realize a particular feature, thereby indicating otherwise hidden



feature dependencies that have evolved over time (intentionally or unintentionally); (2) the *project view* provides a projection of problem reports onto the project directory structure of a system and, as a result, depicts the logical coupling of software parts via one or more features.

These visualizations can be used to point to parts and phases of architectural deterioration on the basis of feature dependencies. Our window of observation was on a yearly scale but that can be adapted to the given data set of the particular system under study. As a result, such visualizations can be used (1) to assess the point of time when some restructuring or reengineering activities should be started, and (2) to estimate the likely amount of resources to be required for changing particular features and/or system parts because of revealed change dependencies.

The work presented is a first step into this research direction. Currently, each validation of visualized indications of architectural deterioration has to be done by an engineer with some external tools. In the end, the visualization and the validation process are envisioned to be integrated in one software evolution analysis environment.

First results using MDS are promising, thus we want to further explore this approach and analyze other large software systems to compare, for instance, the spreading of features in commercial and other open source software. Of further interest are the exploration of higher dimensional solution spaces which should yield more optimized solutions. With *Xgvis* this is quite difficult since the interactive selection and visualization works optimal only on two-dimensional data.

An interesting perspective for future work is the coupling of this visualization approach with architecture recovery tools. One possible direction is to gain insight into the dependencies between problem reports and architectural primitives and styles. More work will be devoted to visualization capabilities such as highlighting or selecting diverse areas for detailed inspection of problem reports. Furthermore, we will investigate the optimization algorithms to allow one to place related features as close to each other as their proportional computed strength indicates.

#### ACKNOWLEDGEMENTS

We thank the Mozilla developers for providing all their data for this case study. Further, we thank the anonymous reviewers for their valuable comments that helped us to improve this paper. The work described in this paper was supported by the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT), The Austrian Industrial Research Promotion Fund (FFF), and the European Commission as EUREKA 2023/ITEA projects CAFE and FAMILIES (see <http://www.infosys.tuwien.ac.at/Cafe/>).

#### REFERENCES

1. Demeyer S, Ducasse S, Nierstrasz O. *Object-oriented Reengineering Patterns*. Morgan Kaufmann Publishers, An Imprint of Elsevier Science: San Francisco CA, USA, 2002; pages 65–94.
2. Fischer M, Pinzger M, Gall H. Analyzing and Relating Bug Report Data for Feature Tracking. *Proceedings 10th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press: Los Alamitos CA, USA, 2003; pages 90–99.
3. Gall H, Krajewski J, Jazayeri M. CVS Release History Data for Detecting Logical Couplings. *Proceedings Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*. IEEE Computer Society Press: Los Alamitos CA, USA, 2003; pages 13–23.
4. Halloran TJ, Scherlis WL. High Quality and Open Source Software Practices, *Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering*, University College Cork, Ireland, 2002. <http://opensource.ucc.ie/icse2002> [5 July 2004].



5. Kemerer CF, Slaughter S. An Empirical Approach to Studying Software Evolution. *IEEE Transactions on Software Engineering*, IEEE Computer Society Press: Los Alamitos CA, USA, 1999; **25**(4):493–509.
6. Zimmermann T, Diehl S, Zeller A. How History Justifies System Architecture (or not). *Proceedings Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*. IEEE Computer Society Press: Los Alamitos CA, USA, 2003; pages 73–83.
7. Fischer M, Pinzger M, Gall H. Populating a Release History Database from Version Control and Bug Tracking Systems. *Proceedings International Conference on Software Maintenance (ICSM'03)*. IEEE Computer Society Press: Los Alamitos CA, USA, 2003; pages 23–32.
8. Kruskal JB, Wish M. Multidimensional Scaling. *Quantitative Applications in the Social Sciences*, Sage Publications: Thousand Oaks CA, USA, 1978; **11**.
9. Taylor CMB, Munro M. Revision Towers. *Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis*. IEEE Computer Society Press: Los Alamitos CA, USA, 2002; pages 43–50.
10. Drahheim D, Pekacki L. Process-Centric Analytical Processing of Version Control Data. *Proceedings Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*. IEEE Computer Society Press: Los Alamitos CA, USA, 2003; pages 131–136.
11. Mockus A, Fielding RT, Herbsleb JD. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM Press: New York NY, USA, 2002; **11**(3):309–346.
12. Bieman J, Andrews A, Yang H. Understanding Change-Proneness in OO Software through Visualization. *Proceedings of 11th International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society Press: Los Alamitos CA, USA, 2003;
13. Gall H, Hajek K, Jazayeri M. Detection of Logical Coupling Based on Product Release History. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, USA, 1998; pages 190–198.
14. Swanson BE. The Dimensions of Maintenance. *Proceedings 2nd International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press: Los Alamitos CA, USA, 1976; pages 492–497.
15. Lanza M. The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques. *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE'04)*. ACM Press: New York NY, USA, 2001; pages 37–42.
16. Lanza M, Ducasse S. Polymetric Views - A Lightweight Visual Approach to Reverse Engineering. *IEEE Transactions on Software Engineering*, IEEE Computer Society Press: Los Alamitos CA, USA, 2003; **29**(9):782–795.
17. Gall H, Jazayeri M, Klösch R, Trausmuth G. Software Evolution Observations Based on Product Release History. *Proceedings 1997 International Conference on Software Maintenance (ICSM '97)*. IEEE Computer Society Press: Los Alamitos CA, USA, 1997; pages 160–166.
18. Gall H, Jazayeri M, Riva C. Visualizing Software Release Histories: The Use of Color and Third Dimension. *Proceedings IEEE International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, USA, 1999; pages 99–108.
19. Zimmermann T, Weißgerber P, Diehl S, Zeller A. Mining Version Histories to Guide Software Changes. *Proceedings 26th International Conference on Software Engineering (ICSE)*. ACM Press: New York NY, USA, 2004; pages 563–572.
20. Dickinson W, Leon D, Podgurski A. Pursuing Failure: The Distribution of Program Failures in a Profile Space. *Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*. ACM Press: New York NY, USA, 2001; pages 246–255.
21. Grune D, Berliner B, Polk J, Klingmon J, Cederqvist P. Version Management with CVS, Signum Support AB, 1992. <http://www.cvshome.org/docs/manual/> [5 April 2004].
22. The Mozilla Organization. Bugzilla Bug Tracking System, Mozilla Foundation: Mountain View CA, USA, 1998–2004. <http://www.bugzilla.org> [5 April 2004].
23. The Mozilla Organization. Mozilla Open-Source Web Browser, Mozilla Foundation: Mountain View CA, USA, 1998–2004. <http://www.mozilla.org> [5 April 2004].
24. The Mozilla Organization. A Bug's Life Cycle, Mozilla Foundation: Mountain View CA, USA, 2002. [http://bugzilla.mozilla.org/bug\\_status.html](http://bugzilla.mozilla.org/bug_status.html) [5 April 2004].
25. Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS. Feature-Oriented Domain Analysis (FODA) Feasibility Study, *Technical Report*, CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University: Pittsburgh, PA, 1990. page 8.
26. Pulvermüller E, Speck A, Coplien JO, D'Hondt M, DeMeuter W. Feature Interaction in Composed Systems. *Proceedings Object-Oriented Technology ECOOP 2001 Workshop Reader*. Springer-Verlag: Heidelberg, Germany, 2001, volume 2323 of *Lecture Notes in Computer Science*; pages 86–97.
27. Wilde N, Gomez JA, Gust T, Strasburg D. Locating User Functionality in Old Code. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, USA, 1992; pages 200–205.



28. Wilde N, Scully MC. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, John Wiley & Sons: Hoboken NJ, USA, 1995; 7(1):49–62.
29. The Free Software Foundation. GNU's Not Unix!, Free Software Foundation: Boston MA, USA, 1996–2004. <http://www.gnu.org> [5 April 2004].
30. Eisenbarth T, Koschke R, Simon D. Aiding Program Comprehension by Static and Dynamic Feature Analysis. *Proceedings IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society Press: Los Alamitos CA, USA, 2001; pages 602–611.
31. Buja A, Swayne DF, Littman M, Dean N, Hofmann H. XGvis: Interactive Data Visualization with Multidimensional Scaling, AT&T Shannon Laboratory, Florham Park, NJ, 2001. <http://www.research.att.com/areas/stat/xgobi/papers/xgvis.pdf> [5 July 2004], <http://www-stat.wharton.upenn.edu/~buja/PAPERS/paper-mds-jcgs.pdf> [5 July 2004].
32. Sutanthavibul S, Smith BV, King P. XFIG Drawing Program for the X Window System, 1985–2002. <http://www.xfig.org> [5 April 2004].
33. Ebert J, Kullbach B, Riediger V, Winter A. GUPRO - Generic Understanding of Programs. *Electronic Notes in Theoretical Computer Science*, Elsevier Science Publishers: Utrecht, Netherlands, 2002; 72(2). <http://www.elsevier.com/jej-ng/31/29/23/127/46/show/Products/notes/index.htm> [5 July 2004].
34. Finnigan P, Holt RC, Kallas I, Kerr S, Kontogiannis K, Müller HA, Mylopoulos J, Perelgut SG, Stanley M, Wong K. The Software Bookshelf. *IBM Systems Journal*, International Business Machines Corporation: Armonk, USA, 1997; 36(4):564–593.
35. Kazman R, Carrière SJ. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Software Engineering*, Kluwer Academic Publishers: Dordrecht, Netherlands, 1999; 6(2):107–138.

#### AUTHORS' BIOGRAPHIES



**Harald Gall** is professor of software engineering at the University of Zurich, Department of Informatics. Prior to that, he was associate professor at the Vienna University of Technology in the Distributed Systems Group (TUV). His research interests are in software engineering with focus on software architectures, reverse engineering, long-term software evolution, program families, as well as distributed and mobile collaboration processes. More information is available at <http://www.ifi.unizh.ch/~gall>



**Michael Fischer** received his master's degree in computer science from the Vienna University of Technology, Austria in 2002. He has contributed to various software projects since 1985. Currently, he is a researcher in the EUREKA/ITEA project FAMILIES focusing on software evolution analysis for program families. More information is available at <http://www.infosys.tuwien.ac.at/staff/mf/>